

UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

Lab1: Experimental setup and tools



6th March 2019, Q1

Contents

1	Introduction	2
2	Experimental setup	2
2.1	Node architecture and memory	2
2.2	Serial compilation and execution	3
2.3	Compilation and execution of OpenMP programs	3
3	Strong vs. weak scalability	3
3.1	Strong scalability	3
3.1.1	Boada 1-4	4
3.1.2	Boada 5	4
3.1.3	Boada 6-8	5
3.2	Weak scalability	6
3.2.1	Boada 1-4	6
3.2.2	Boada 5	6
3.2.3	Boada 6-8	6
4	Analysis of task decompositions for 3DFFT	7
4.1	Original Version	7
4.2	Version 1	9
4.3	Version 2	11
4.4	Version 3	14
4.5	Version 4	16
4.6	Version 5	18
4.7	Analysis of the potential strong scalability in v4 and v5	22
4.8	Summary Table	25
5	Understanding the parallel execution of 3DFFT	26
5.1	Initial version	26
5.2	Improving ϕ	29
5.3	Reducing parallelisation overheads	31
5.4	Conclusions	33

1 Introduction

2 Experimental setup

2.1 Node architecture and memory

In this first part of Laboratory 1, we had to search among the boada nodes to investigate the characteristics of each one of them. To do this we have had to use the commands **lscpu** and **lstopo**. We have obtained the following data:

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395 MHz	2600 MHz	1700 MHz
L1-I cache size (per-core)	32 kB	32 kB	32 kB
L1-D cache size (per-core)	32 kB	32 kB	32 kB
L2 cache size (per-core)	256 kB	256 kB	256 kB
Last-level cache size (per-socket)	12288 kB	15360 kB	20480 kB
Main memory size (per socket)	12 GB	31 GB	16 GB
Main memory size (per node)	23 GB	63 GB	31 GB

Table 1: Architectural characteristics of the different node types available in boada.

This information has been very useful to compare side-by-side the different node types in boada. Moreover, the figures generated using **lstopo** were also useful to have a better idea of the architecture of each one graphically.

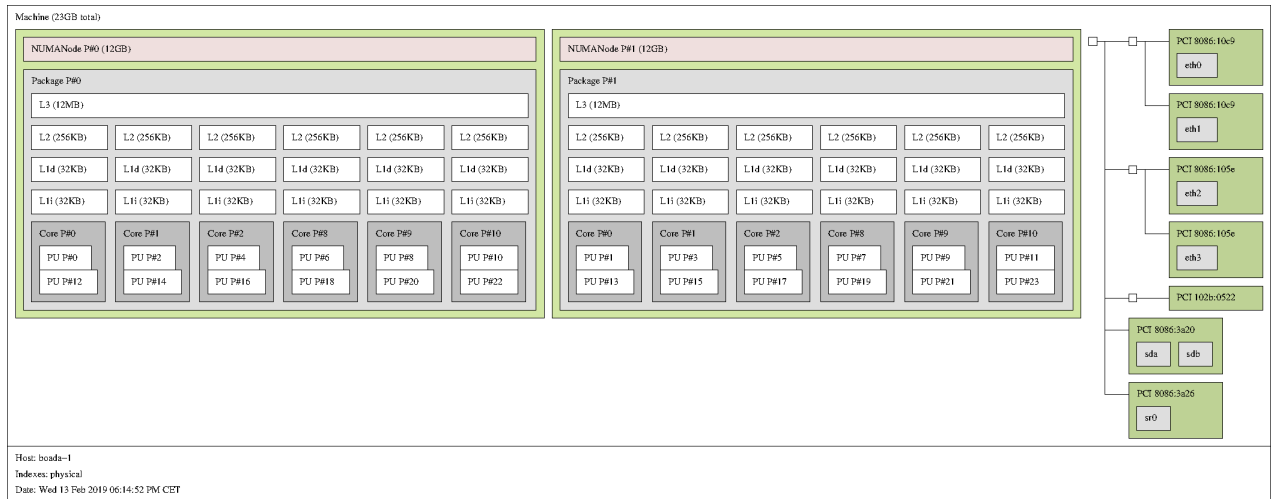


Figure 1: Architectural diagram for the node **boada-1**, obtained using the **lstopo** command.

2.2 Serial compilation and execution

In this part, we have worked with a code used to find an approximation of the pi number. We have used this code to test the performance of the executable in the different nodes of boada. There is a big difference executing the code in one node or in another. In node 1, it is executed interactively (since everyone is connected at the same time in this node) whereas in the other nodes, we will achieve an isolated execution ensuring the maximum performance of the node. We have obtained the following results:

Execution time in node 1: 3.939644 s

Execution time in node 4: 3.958794 s

Theoretically the execution time in node 4 should be smaller than in node 1 but we have supposed that the status of the machine (maybe too heat) can cause this difference.

2.3 Compilation and execution of OpenMP programs

In this part of the laboratory we made the first contact with an OpenMP code, where we have been able to observe different instructions used to carry out with the parallelization. Executing the code we have obtained the following results interactively in the node 1:

Execution time with 1 thread: 3.944351 s

Execution time with 8 thread: 0.581683 s

And then the same code has been executed isolated in the node 3:

Execution time with 1 thread: 3.944042 s

Execution time with 8 thread: 0.943658 s

As we can see, the execution in node 1 is again smaller than in node 3 for the same reason we said before.

3 Strong vs. weak scalability

In this section we had to explore the scalability of the parallel version in **pi_omp.c**. This scalability is measured calculating the speedup. This is the ratio between the sequential and the parallel execution times. To do this, we had 2 different scenarios: *Strong scalability* and *Weak scalability*.

3.1 Strong scalability

Strong scalability is a technique where the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of

the program.

We are going to compare the execution time of **pi_omp.c** in the 3 different node architectures. We will see that is not always a good solution to increase the number of threads as much as we can, because it will appear a lot of overhead time.

3.1.1 Boada 1-4

As we can see in Figure 2 the execution time decreases as we increase the number of threads, reaching an optimum point in 11 threads. Then, it begins to increase again. This may be because of the overhead times when creating tasks, synchronizing and asking for data between them.

In the second figure, on the other hand, we can observe the real speedup (continuous line) and the teoric speedup (dashed line). The ideal case would be when the real speedup is coincident with the teoric speedup. This is not the case for the reason we said before (time overheads). We see a maximum speedup when the number of threads is 11. This is when the execution time is the minimum.

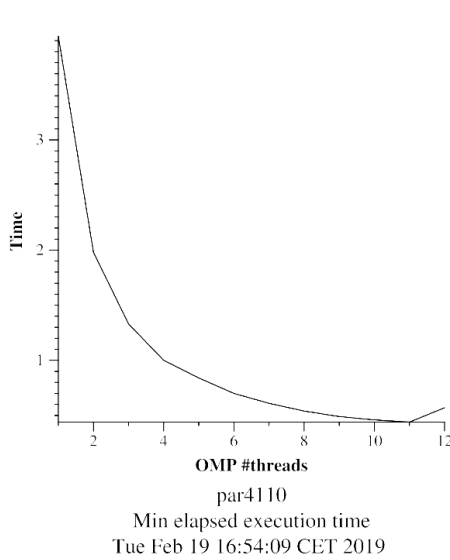


Figure 2: Execution time plot varying the number of threads in boada 1-4.

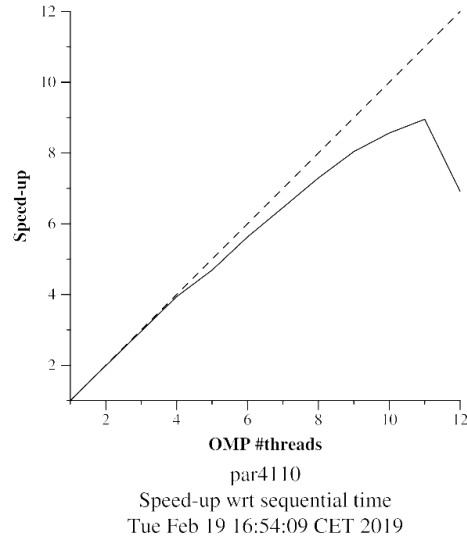


Figure 3: Speedup plot varying the number of threads in boada 1-4.

3.1.2 Boada 5

We can see that Figure 4 is very similar than Figure 2 but with 12 threads it has not reached its minimum execution time. So we think that adding more threads would still decrease the time.

However, there is something rare in Figure 5, because the real speedup is bigger than the teoric speedup, when the real speedup should never pass the teoric one. This means that the sequential version is faster than the parallel one. We discussed about it and we came to a conclusion: this happens because sequential code may

have more hits in the cache than the parallel version, so it spends less time in memory accesses.

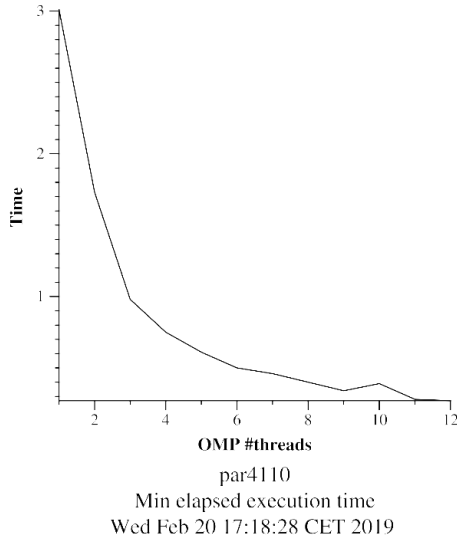


Figure 4: Execution time plot varying the number of threads in boada 5.

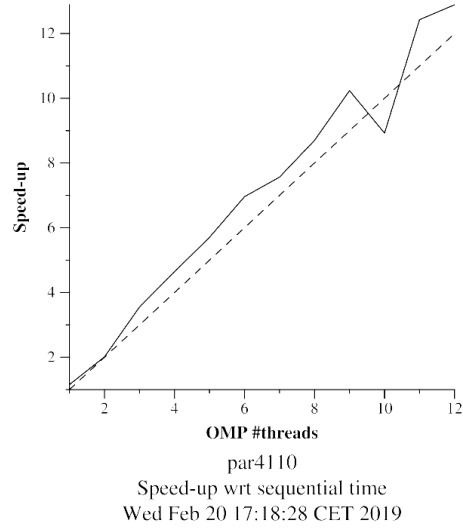


Figure 5: Speedup plot varying the number of threads in boada 5.

3.1.3 Boada 6-8

In this point, all important things have been explained before. The only difference is that this architecture seems to be better than the rest, with a speedup that increases linear and is coincident with the teoric one. In this nodes there are also time overheads, but as it is a newer version, this overheads may be treated better. If we do the same plot using a bigger number of threads, the real speedup will decay for sure and will become a similar plot from the previous ones.

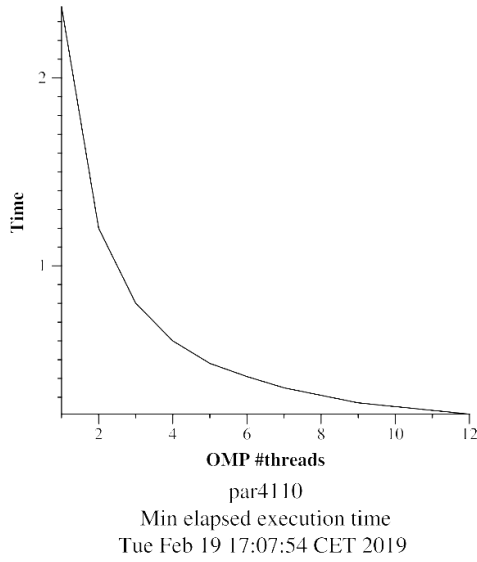


Figure 6: Execution time plot varying the number of threads in boada 6-8.

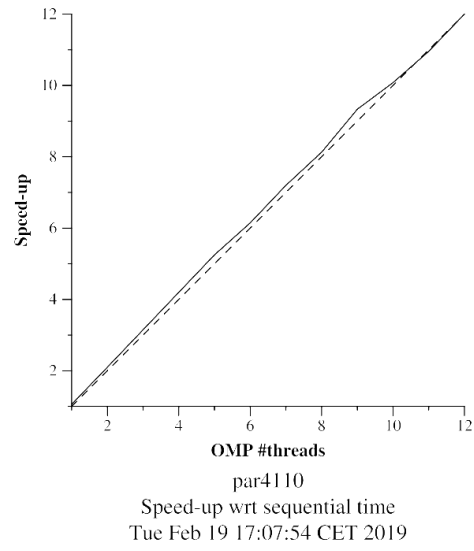


Figure 7: Speedup plot varying the number of threads in boada 6-8.

To conclude, boada 6-8 seems to be the best architecture of the 3 we have in strong scalability because it is the one where the real speedup is more time coincident with the teoric speedup.

3.2 Weak scalability

In weak scalability, the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size for which the program is executed.

3.2.1 Boada 1-4

In Figure 8, the teoric speedup would be a horizontal line in $y = 1.0$ but this does not happen for the reason we said before, the overhead times. The abrupt decrease is found from 9 to 11 threads, when t_{create} , t_{sync} and t_{access} begin to be relevant in the execution time.

3.2.2 Boada 5

In Figure 9 we can observe again that the real speedup is greater than the teoric speedup due to cache hits and misses, as we said in the previous section.

3.2.3 Boada 6-8

Finally, in boada 6-8 we see that the speedup stays quite good with respect to the teoric one. It only decreases at the begining. Then, it stays constant in $y \approx 0.96$, which is very good.

Again, we can conclude that boada 6-8 architecture is the best solution in weak scalability.

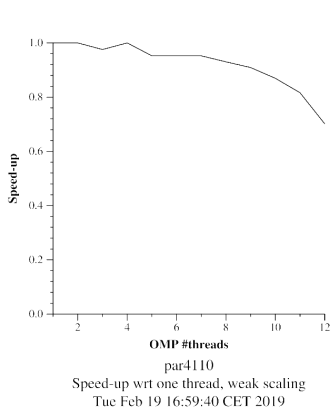


Figure 8: Execution time plot varying the number of threads and the problem size in boada 1-4.

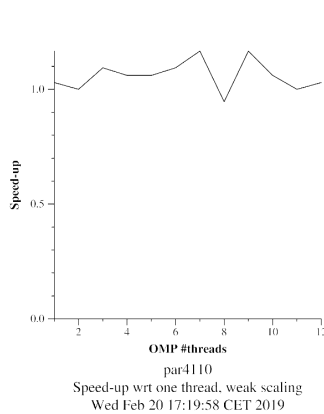


Figure 9: Execution time plot varying the number of threads and the problem size in boada 5.

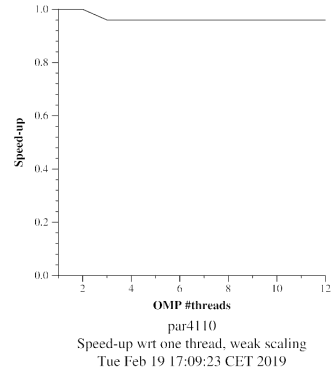


Figure 10: Execution time plot varying the number of threads and the problem size in boada 6-8.

4 Analysis of task decompositions for 3DFFT

In order to do the analysis we have used *Tareador*, which helped us to see the task dependence graph and analyse the variables whose access provokes each data dependence between a pair of nodes (tasks). Moreover we have also used *Paraver* that showed the timeline for the simulated execution using the number of processors that we wanted. Even though the maximum number of processors that *Paraver* could simulate was 128, we never had to use all of them to get the maximum parallelism of the code. However, it was useful to know what the T_∞ was.

4.1 Original Version

In this version, there were few parts of the code parallelised, so parallelism will be small.

To do the calculations manually, we need the time of execution per instruction. To do this, we need the total number of instructions and the T_1 . Afterwards, we only have to divide the number of instructions by the T_1 . Thus, $T_{instruction} = 639780001/639780 \approx 1000$ ns.

Using $T_{instruction}$ we can now calculate the data we want. On the one hand, T_1 refers to the execution time of the code in sequential mode (using only 1 processor). Hence, we have to multiply $T_{instruction}$ by the total number of instructions. On the other hand, T_∞ refers to the execution time of the code when we have an infinite number of processors. T_∞ is calculated finding the critical path in the task dependency graph. This is the largest path that cannot be parallelised. Then, we multiply the number of instructions in the critical path by $T_{instruction}$.

$$T_1 = 639780 * 1000 = 639780000ns$$

$$T_\infty = 639760 * 1000 = 639760000ns$$

As we can see, T_1 and T_∞ values are very similar from the ones of the *Tareador*.

Finally, we had to compute the Parallelism, which refers to how fast would we go if sufficient (infinite) resources were available. For computing it we divided T_1 by T_∞ .

$$Parallelism = 639780000/639760000 = 1.00003126172$$

The following code is the one extracted from the original version. We can easily see that this code can be decomposed into finer-grained tasks. We will see how we decompose it in the following versions. Figure 11 reflects the need we have to decompose the tasks to reduce the execution time and increase the parallelism, as task *init_complex_grid* (red task) has 100442 instructions and *ffts1_and_transpositions* (yellow task) has 539208 instructions.

```
START_COUNT.TIME;

tareador_start_task("Seq code");

ffts1_planes(p1d, in_fftw);
transpose_xy_planes(tmp_fftw, in_fftw);
ffts1_planes(p1d, tmp_fftw);
transpose_zx_planes(in_fftw, tmp_fftw);
ffts1_planes(p1d, in_fftw);
transpose_zx_planes(tmp_fftw, in_fftw);
transpose_xy_planes(in_fftw, tmp_fftw);

tareador_end_task("Seq code");

STOP_COUNT.TIME("Execution FFT3D");

...
```

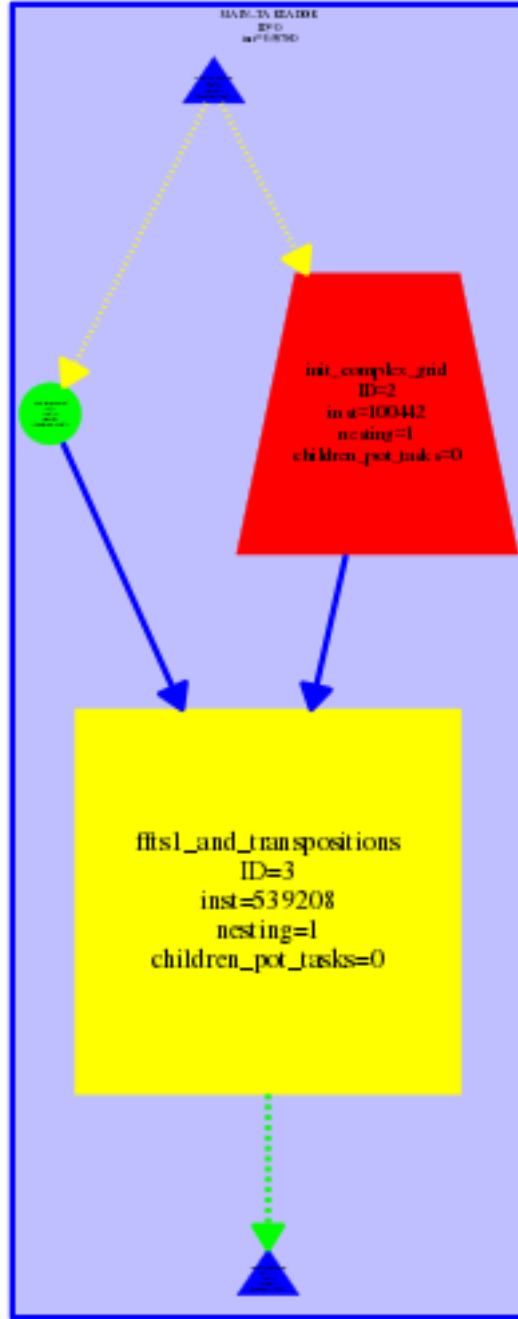


Figure 11: Task dependency graph of the original version.

4.2 Version 1

From now on, we are not going to talk again on how to compute T_∞ and Parallelism because it is the same explained in the previous section. Moreover, $T_{instruction}$ and T_1 will remain the same values due to the fact that we are executing the code in the same boada node and the code lines are always the same.

$$T_\infty = (110+100442+103144+57444+103144+57444+103144+57444+57444+0)*1000$$

$$= 639760 * 1000 = 639760000ns$$

$$Parallelism = T_1/T_\infty = 639780000/639760000 = 1.000031262$$

In this first version of the code we created one task for each function that was inside the "big" task of the original code. As a consequence, the work that was doing just one processor could now be done by 7 processors at the same time. This is not true because of the dependencies that are created. This can be seen in Figure 12. The yellow task cannot be executed until the red and green ones end, the purple task cannot be executed until the yellow task terminates, and so on. For this reason, the Parallelism is approximately 1 because, even though we have decomposed the tasks, only 1 big task can be executed at the same time, although we have had an infinite number of processors.

```
START_COUNT_TIME;

tareador_start_task("ffts1_and_transpositions_1");
    ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions_1");

tareador_start_task("ffts1_and_transpositions_2");
    transpose_xy_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions_2");

tareador_start_task("ffts1_and_transpositions_3");
    ffts1_planes(p1d, tmp_fftw);
tareador_end_task("ffts1_and_transpositions_3");

tareador_start_task("ffts1_and_transpositions_4");
    transpose_zx_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions_4");

tareador_start_task("ffts1_and_transpositions_5");
    ffts1_planes(p1d, in_fftw);
tareador_end_task("ffts1_and_transpositions_5");

tareador_start_task("ffts1_and_transpositions_6");
    transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1_and_transpositions_6");

tareador_start_task("ffts1_and_transpositions_7");
    transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1_and_transpositions_7");

STOP_COUNT_TIME("Execution FFT3D");

...
```

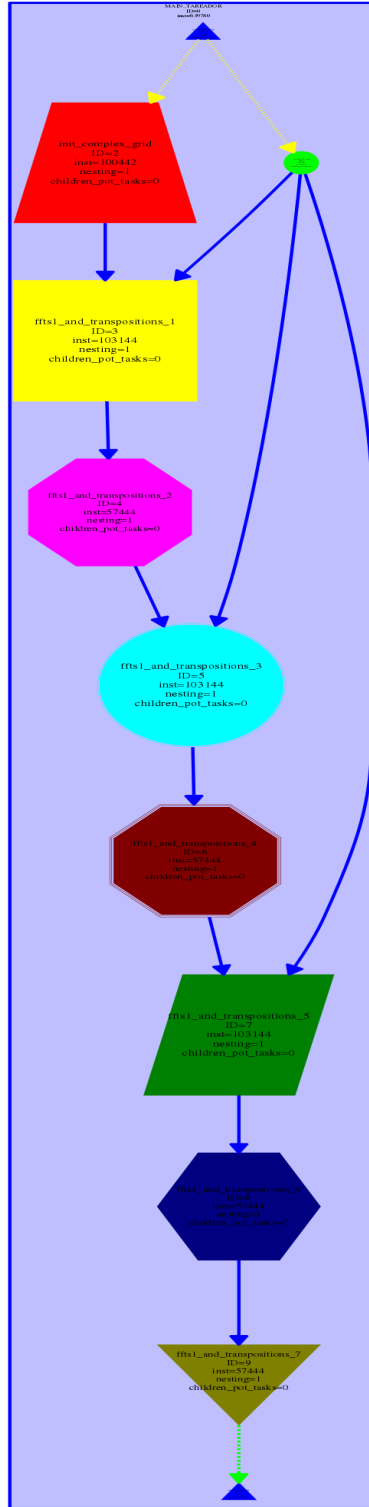


Figure 12: Task dependency graph of version 1.

4.3 Version 2

We have seen in version 1 that creating a task each time we call a function is not a good decision at all due to the dependencies. However, if we create the tasks inside

the first for of some functions, we will be able to parallelise more the code because for every iteration of the loop we will create another task that can be executed in a different processor.

In this version, we will only modify the function *ffts1_planes*. We will create one task for every iteration of the most outer loop of the function. Thus, we will create N different tasks every time we call this function. The code below shows how we have done it.

$$T_{\infty} = (57 + 100442 + 10305 + 57444 + 10305 + 57444 + 10305 + 57444 + 57444 + 0) * 1000$$

$$= 361190 * 1000 = 361190000ns$$

$$Parallelism = T_1/T_{\infty} = 639780000/361190000 = 1.771311498$$

We can see that now the parallelism has increased a lot in comparison to the one in version 1 and the critical path has decreased a lot.

```
void ffts1_planes(fftwf_plan pld, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
        tareador_start_task("ffts1_planes_loop_k");
        for (j=0; j<N; j++) {
            fftwf_execute_dft(pld, (fftwf_complex *)in_fftw[k][j][0],
                             (fftwf_complex *)in_fftw[k][j][0]);
        }
        tareador_end_task("ffts1_planes_loop_k");
    }
}

...

int main(){
    ...

    START_COUNT.TIME;

    ffts1_planes(pld, in_fftw);

    tareador_start_task("ffts1_and_transpositions_2");
        transpose_xy_planes(tmp_fftw, in_fftw);
    tareador_end_task("ffts1_and_transpositions_2");

    ffts1_planes(pld, tmp_fftw);

    tareador_start_task("ffts1_and_transpositions_4");
        transpose_zx_planes(in_fftw, tmp_fftw);
    tareador_end_task("ffts1_and_transpositions_4");

    ffts1_planes(pld, in_fftw);
```

```

tareador_start_task("ffts1-and-transpositions_6");
    transpose_zx_planes(tmp_fftw, in_fftw);
tareador_end_task("ffts1-and-transpositions_6");

tareador_start_task("ffts1-and-transpositions_7");
    transpose_xy_planes(in_fftw, tmp_fftw);
tareador_end_task("ffts1-and-transpositions_7");

STOP_COUNT.TIME("Execution FFT3D");

...
}

```

In Figure 13 we can see that the yellow task of Figure 12 has been decomposed into smaller tasks (the yellow ones too). Now, up to 10 processors can execute these diferent tasks at the same time, because there is no dependencies between them.

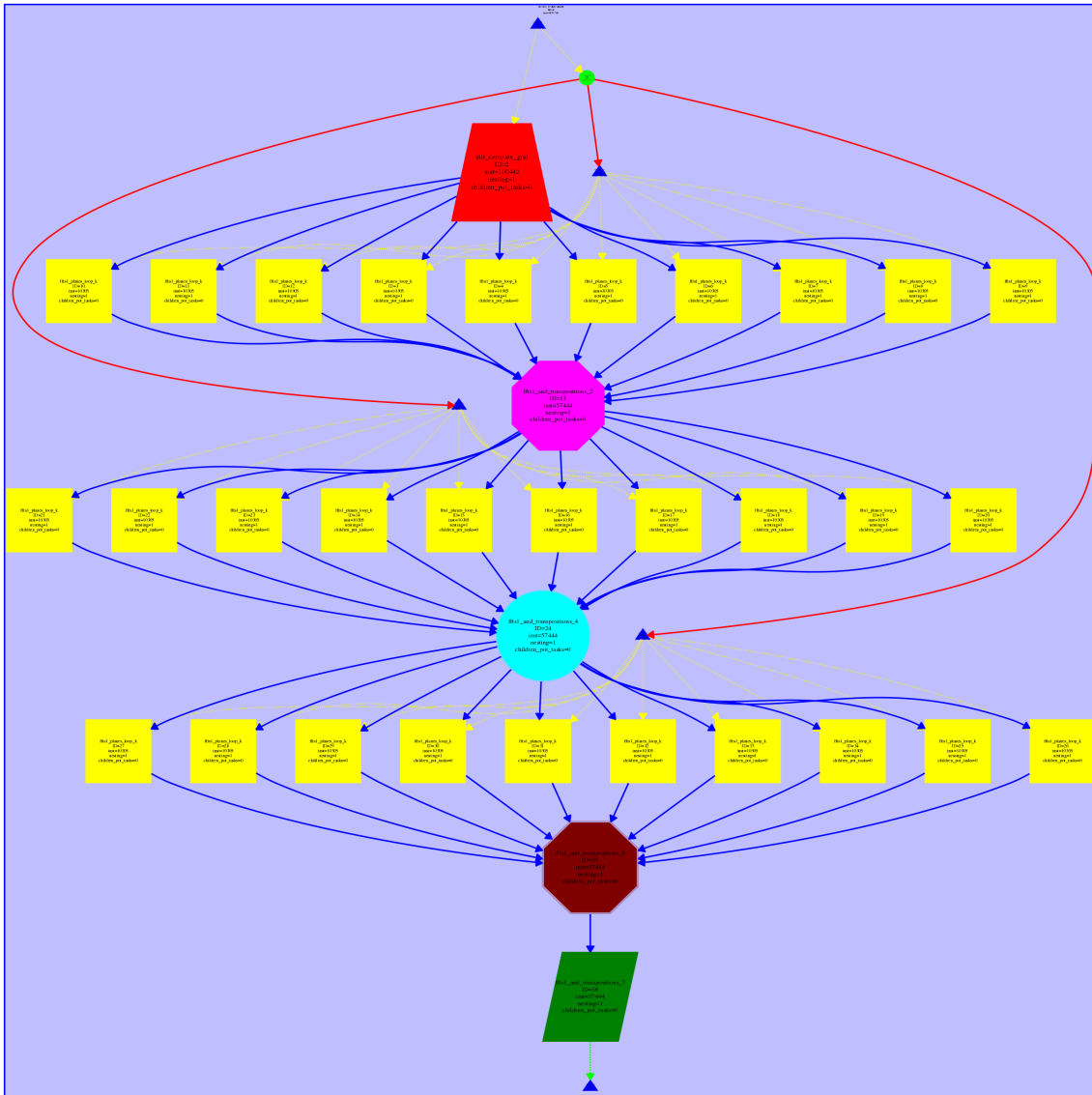


Figure 13: Task dependency graph of version 2.

4.4 Version 3

As we saw that the modification in version 2 improved the parallelism and decreased T_∞ , we continued "on the same line", so we started modifying *transpose_xy_planes* and *transpose_zx_planes* using the same strategy than in version 2. This is creating the tasks inside the functions, specifically inside the most outer loop. Doing this we got the following results:

$$T_\infty = (57 + 100442 + 10305 + 5735 + 10305 + 5735 + 10305 + 5735 + 5375 + 0) * 1000$$

$$= 154354 * 1000 = 154354000ns$$

$$Parallelism = T_1/T_\infty = 639780000/154354000 = 4.144887726$$

In this version, Parallelism has increased much more than in version 2. In Figure 14 we can see that the purple and light blue tasks of Figure 13 have now become 10 tasks respectively (these 2 loop also depend on the N value).

```
void transpose_xy_planes(fftwf_complex tmp_fftw[][N][N],
                        fftwf_complex in_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
        tareador_start_task("ffts1_transpose_xy_loop_k");
        for (j=0; j<N; j++) {
            for (i=0; i<N; i++)
            {
                tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
                tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
            }
        }
        tareador_end_task("ffts1_transpose_xy_loop_k");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N],
                        fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
        tareador_start_task("ffts1_transpose_zx_loop_k");
        for (j=0; j<N; j++) {
            for (i=0; i<N; i++)
            {
                in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
                in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
            }
        }
        tareador_end_task("ffts1_transpose_zx_loop_k");
    }
}
```

```

...

int main() {

    ...

    START_COUNT_TIME;

    tareador_start_task("init_complex_grid");
    init_complex_grid(in_fftw);
    tareador_end_task("init_complex_grid");

    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
    ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
    ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
    transpose_xy_planes(in_fftw, tmp_fftw);

    STOP_COUNT_TIME("Execution FFT3D");

    ...
}

```

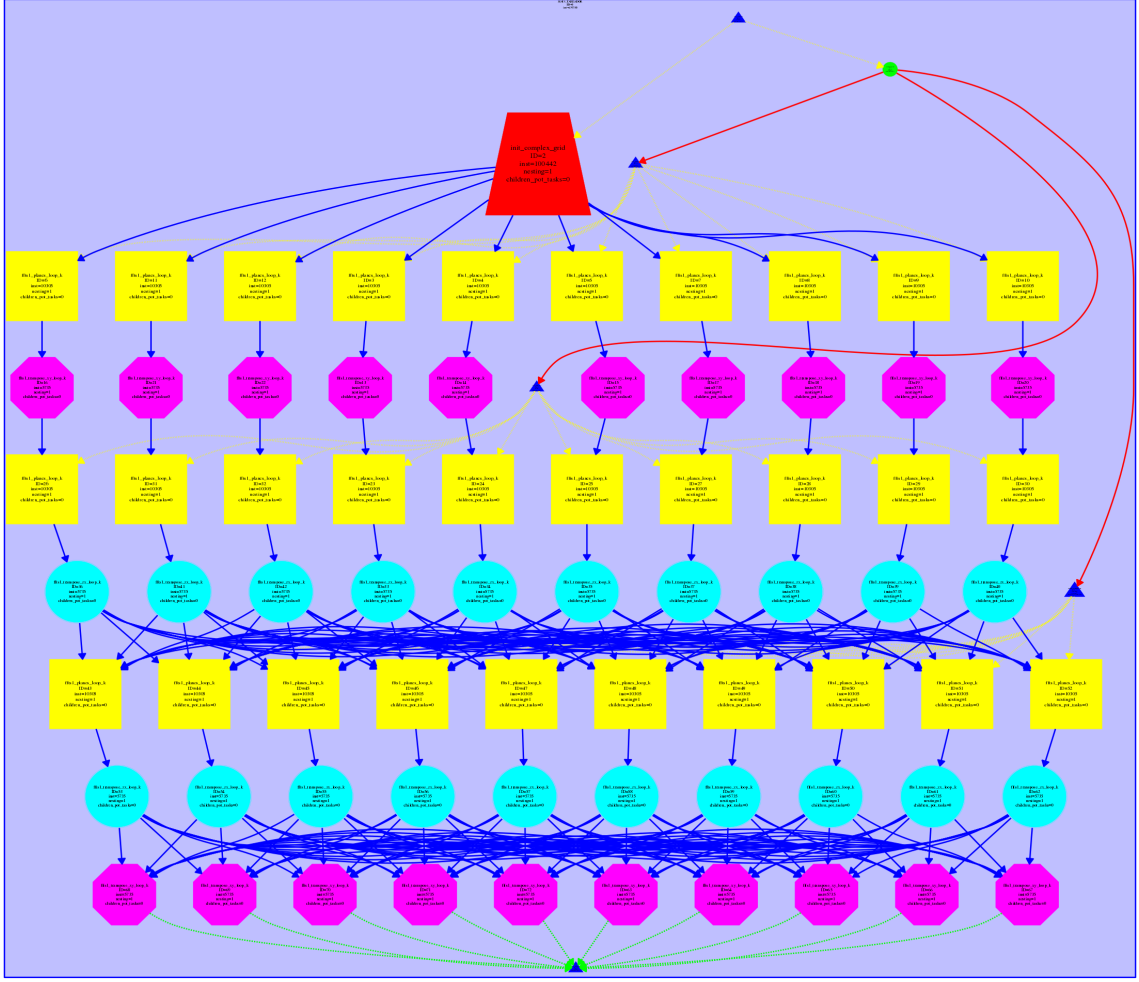



Figure 14: Task dependency graph of version 3.

4.5 Version 4

Even though in version 3 we improved a lot the code, in the previous figure we can see that there is still one task, the red one, that has 100442 instructions inside him, so we can also decompose this task into smaller ones doing the same modifications than the previous 2 versions.

We have modified the *init_complex_grid* function, creating a new task every time we iterate through the most outer loop of the function. We can see the modifications graphically in Figure 15 (red tasks). Again, there are N new tasks.

$$T_{\infty} = (128 + 10035 + 10305 + 5735 + 10305 + 5735 + 10305 + 5735 + 5375 + 0) * 1000$$

$$= 64018 * 1000 = 64018000ns$$

$$Parallelism = T_1/T_{\infty} = 639780000/64018000 = 9.993751757$$

In this version, the parallelism has increased a lot with respect to version 3.

We also had to simulate this version of the code with 1, 2, 4, 8, 16 and 32 processors to analyse the potential strong scalability. This is discussed in the section 4.7.

```

void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

    for (k = 0; k < N; k++) {
        tareador_start_task("ffts1_complex_loop_k");
        for (j = 0; j < N; j++) {
            for (i = 0; i < N; i++)
            {
                in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+
                                                sin(M_PI*((float)i)/32.0)+
                                                sin(M_PI*((float)i)/16.0));
                in_fftw[k][j][i][1] = 0;
#ifdef TEST
                out_fftw[k][j][i][0] = in_fftw[k][j][i][0];
                out_fftw[k][j][i][1] = in_fftw[k][j][i][1];
#endif
            }
        }
        tareador_end_task("ffts1_complex_loop_k");
    }
}

...

int main() {
    ...

    START_COUNT_TIME;

    tareador_start_task("start_plan_forward");
    start_plan_forward(in_fftw, &p1d);
    tareador_end_task("start_plan_forward");

    STOP_COUNT_TIME("3D FFT Plan Generation");

    START_COUNT_TIME;

    init_complex_grid(in_fftw);

    STOP_COUNT_TIME("Init Complex Grid FFT3D");

    START_COUNT_TIME;

    ffts1_planes(p1d, in_fftw);
    transpose_xy_planes(tmp_fftw, in_fftw);
    ffts1_planes(p1d, tmp_fftw);
    transpose_zx_planes(in_fftw, tmp_fftw);
    ffts1_planes(p1d, in_fftw);
    transpose_zx_planes(tmp_fftw, in_fftw);
}

```

```

transpose_xy_planes(in_fftw , tmp_fftw);

STOP_COUNT.TIME("Execution FFT3D");

...
}

```

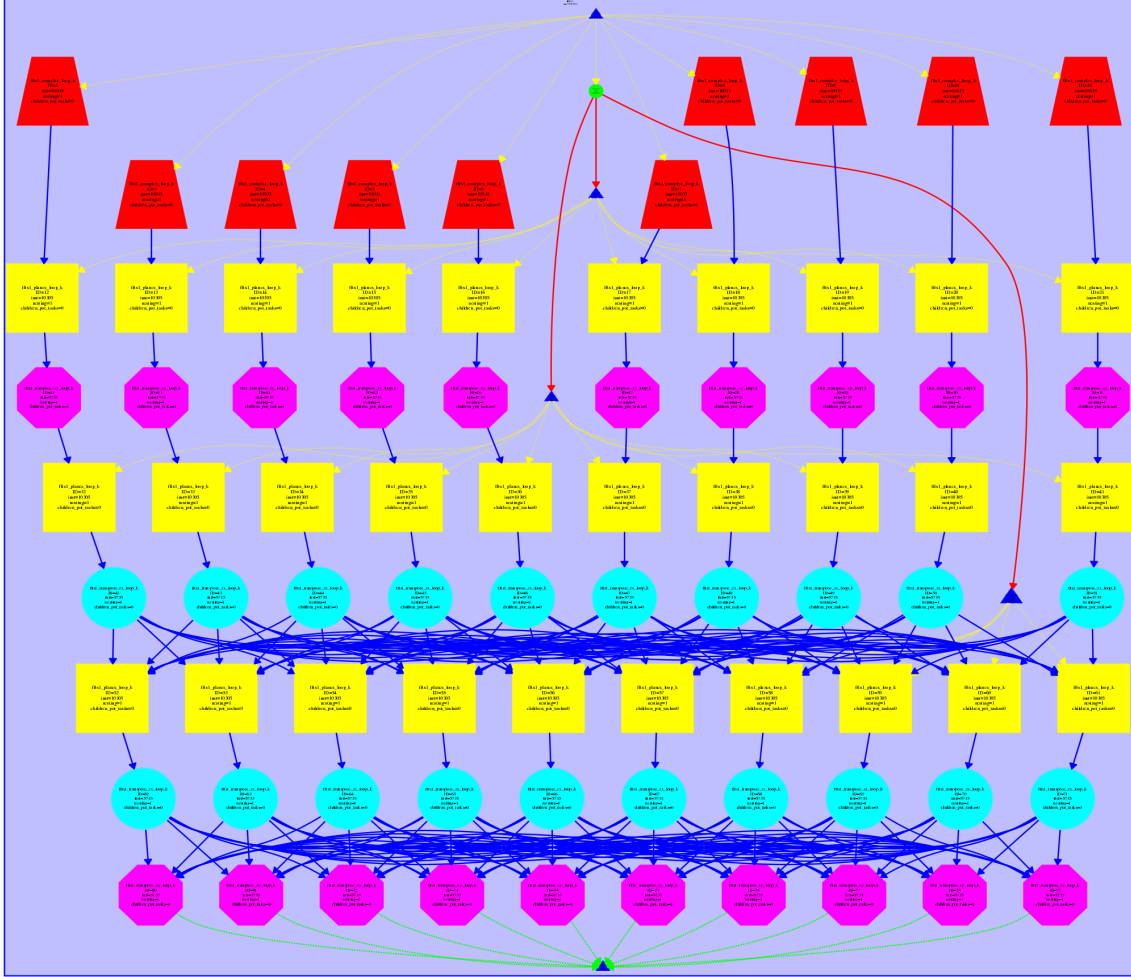


Figure 15: Task dependency graph of version 4.

4.6 Version 5

Finally, in the last version we had to decompose into smaller tasks the ones that we thought that were necessary and useful to decrease T_∞ and increase Parallelism. What we did was to look in the Figure 15 the tasks that had a bigger number of instructions. There were 2 functions with the same number of instructions, 10305. These two functions were *init_complex_grid* and *ffts1_planes*.

We "sunk" the creation of the tasks one more loop, so now in *init_complex_grid* and *ffts1_planes* we create N^2 tasks, even though in the first one we could have created N^3 tasks. We did not that because we thought that it could cause a lot of overheads and this was not good.

$$T_1 = (970 + 20 + 1065 + 1022 + 5735 + 1022 + 5735 + 1022 + 5735 + 5735 + 0) * 1000$$

$$= 28061 * 1000 = 28061000ns$$

$$Parallelism = T_1/T_\infty = 639780000/28061000 = 22.79961512$$

In this last version, we got the biggest parallelism of all versions and the smallest critical version. As in version 4, we also had to simulate this version of the code with 1, 2, 4, 8, 16 and 32 processors to analyse the potential strong scalability. This is discussed in the section 4.7.

```

void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

    for (k = 0; k < N; k++) {
        for (j = 0; j < N; j++) {
            tareador_start_task("ffts1_complex_loop_kj");
            for (i = 0; i < N; i++)
            {
                in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+
                                                sin(M_PI*((float)i)/32.0)+
                                                sin(M_PI*((float)i)/16.0));
                in_fftw[k][j][i][1] = 0;
            }
            #if TEST
                out_fftw[k][j][i][0] = in_fftw[k][j][i][0];
                out_fftw[k][j][i][1] = in_fftw[k][j][i][1];
            #endif
        }
        tareador_end_task("ffts1_complex_loop_kj");
    }
}

void ffts1_planes(fftwf_plan pld, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
        for (j=0; j<N; j++) {
            tareador_start_task("ffts1_planes_loop_kj");
            fftwf_execute_dft(pld, (fftwf_complex *)in_fftw[k][j][0],
                             (fftwf_complex *)in_fftw[k][j][0]);
            tareador_end_task("ffts1_planes_loop_kj");
        }
    }
}

...

int main() {
    ...
}

```

```

START_COUNT.TIME;

tareador_start_task("start_plan_forward");
start_plan_forward(in_fftw, &p1d);
tareador_end_task("start_plan_forward");

STOP_COUNT.TIME("3D FFT Plan Generation");

START_COUNT.TIME;

init_complex_grid(in_fftw);

STOP_COUNT.TIME("Init Complex Grid FFT3D");

START_COUNT.TIME;

ffts1_planes(p1d, in_fftw);
transpose_xy_planes(tmp_fftw, in_fftw);
ffts1_planes(p1d, tmp_fftw);
transpose_zx_planes(in_fftw, tmp_fftw);
ffts1_planes(p1d, in_fftw);
transpose_zx_planes(tmp_fftw, in_fftw);
transpose_xy_planes(in_fftw, tmp_fftw);

STOP_COUNT.TIME("Execution FFT3D");

...
}

```

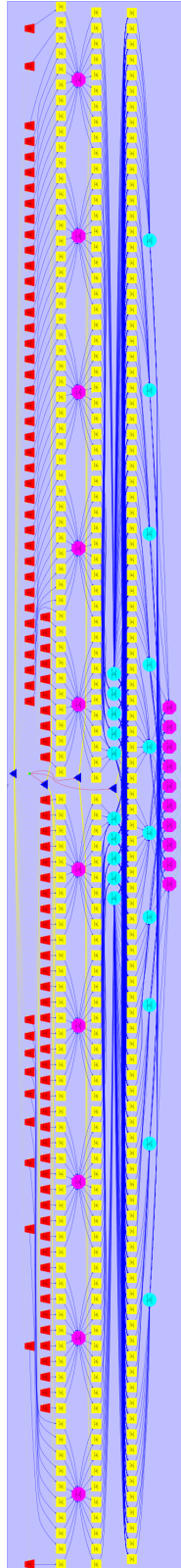


Figure 16: Task dependency graph of version 5.

4.7 Analysis of the potential strong scalability in v4 and v5

In v4 and v5 of **3dfft.tar.c** we had to analyse the potential strong scalability they had. As we said in the first laboratory session, in *strong* scalability the number of processors is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of the program. We will analyse it using 1, 2, 4, 8, 16 and 32 processors. *Paraver* is very useful in it because it can simulate up to 128 processors and show the T_p of the program. Firstly we will talk about v4 plots and secondly about v5 plots.

As we can see in Figure 17, the bigger the number of CPUs, the faster the program runs, achieving a minimum time of execution from 16 processors. This is because the new processors will not execute any instruction. Besides, the Figure 18 shows the speedup. Speedup is the relative reduction of execution time when using P processors with respect to sequential (T_1/T_p). As we said before, we reduced the execution time as we were increasing the number of processors, until 16. For this reason, the speedup increases also until 16 processors.

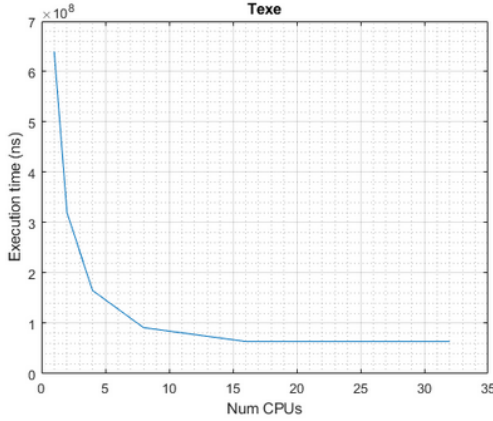


Figure 17: Execution time plot of version 4 as we increase the number of processors.

Processors	T_{exe} [ns]
1	639780001
2	320310001
4	165389001
8	91496001
16	64018001
32	64018001

Table 2: Values for Figure 17.

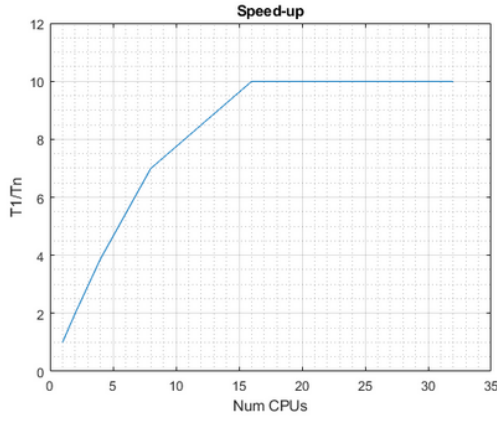


Figure 18: Speedup plot of version 4 as we increase the number of processors.

Processors	Speed-up
1	1
2	1.9973775
4	3.8683346
8	6.9924368
16	9.9937516
32	9.9937516

Table 3: Values for Figure 18.

If we look now in Figures 19 and 20, we can see very similar plots from the ones explained before. The main difference is that in this two plots, we have not reached a minimum execution time, so simulating the code with more than 32 processors would reduce the time. As a consequence, speedup also has not reached its maximum value, and adding more processors would still increase its value.

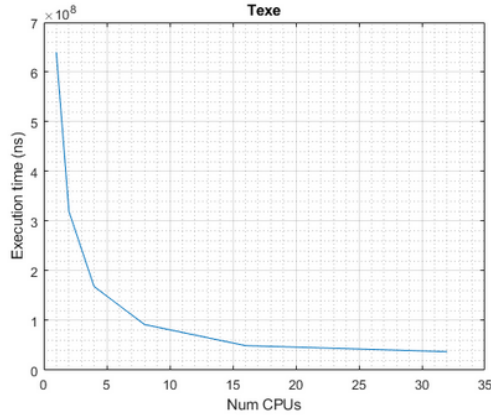


Figure 19: Execution time plot of version 5 as we increase the number of processors.

Processors	T_{exe} [ns]
1	639780001
2	320020001
4	168343001
8	92001001
16	49520001
32	37337001

Table 4: Values for Figure 19.

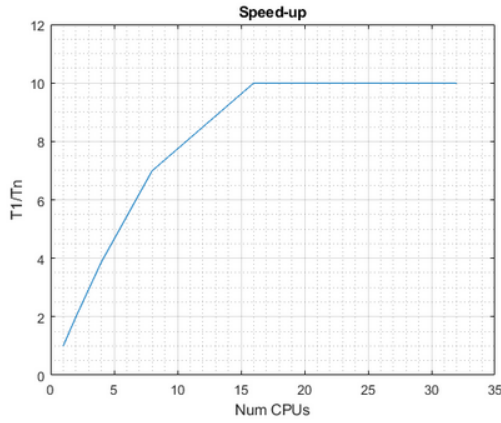


Figure 20: Speedup plot of version 5 as we increase the number of processors.

Processors	Speed-up
1	1
2	1.9991875
4	3.8004550
8	6.9540548
16	12.919628
32	17.135281

Table 5: Values for Figure 20.

Figures 21 and 22 show the main differences of the code between v4 and v5. These are "compact" codes. For more information see the codes of the sections Version 4 and Version 5.

We could easily see that in Figure 15 tasks *ffts1_complex_loop_k* and *ffts1_planes_loop_k* were the ones with more number of instructions, so these were the ones we had to decompose into finer-grained tasks. For this reason, in *init_complex_grid* function the tasks are now created inside the second loop unlike v4, that were created within the first loop. We could have created smaller tasks if we had created the tasks inside the most inner loop.

On the other hand, in *ffts1_planes* function we now create the tasks inside the second loop.

These two changes have increased a lot the parallelism.

```

void init_complex_grid (...) {
    int k,j,i;

    for (k = 0; k < N; k++) {
        tareador_start_task("v4_complex");
        for (j = 0; j < N; j++) {
            for (i = 0; i < N; i++)
            {
                ...
            }
        }
        tareador_end_task("v4_complex");
    }
}

void ffts1_planes (...) {
    int k,j;

    for (k=0; k<N; k++) {
        tareador_start_task("v4_planes");
        for (j=0; j<N; j++) {
            ...
        }
        tareador_end_task("v4_planes");
    }
}

```

Figure 21: Compact code of version 4.

```

void init_complex_grid (...) {
    int k,j,i;

    for (k = 0; k < N; k++) {
        for (j = 0; j < N; j++) {
            tareador_start_task("v5_complex");
            for (i = 0; i < N; i++)
            {
                ...
            }
            tareador_end_task("v5_complex");
        }
    }
}

void ffts1_planes (...) {
    int k,j;

    for (k=0; k<N; k++) {
        for (j=0; j<N; j++) {
            tareador_start_task("v5_planes");
            ...
            tareador_end_task("v5_planes");
        }
    }
}

```

Figure 22: Compact code of version 5.

4.8 Summary Table

To conclude with this section, table 6 summarizes all the information extracted from *Tareador*, *Paraver* and the manual calculations.

T_1 using *Tareador* was calculated simulating the execution with 1 processor and T_∞ simulating the execution with the maximum number of processors that *Paraver* can, 128.

Version		T_1 [ns]	T_∞ [ns]	Parallelism
seq	Manual	639780000	639760000	1.000031262
	Tareador	639780001	639760001	1.000031262
v1	Manual	639780000	639760000	1.000031262
	Tareador	639780001	639707001	1.000114115
v2	Manual	639780000	361190000	1.771311498
	Tareador	639780001	361190001	1.771311496
v3	Manual	639780000	154354000	4.144887726
	Tareador	639780001	154354001	4.144887705
v4	Manual	639780000	64018001	9.993751601
	Tareador	639780001	64018001	9.993751601
v5	Manual	639780000	28061000	22.79961512
	Tareador	639780001	27971001	22.87297480

Table 6: Analysis of task decompositions for the 3dfft program.

5 Understanding the parallel execution of 3DFFT

In this final session, we had to work with the *Paraver* environment, used to gather information about the execution of a parallel application in *OpenMP* and visualize it.

We had to calculate several things:

- T_1 : This is the execution time using only 1 processor (sequential). We can get this value from the *Paraver*.
- T_8 : This is the execution time using 8 processors at the same time. This value can also be extracted from *Paraver*, simulating the code with 8 CPUs.
- S_8 : The speedup we have using 8 processors. We can obtain this value dividing T_1 by T_8 .
- ϕ : Represents the amount of parallelised code with respect to the total.
- S_∞ : Speedup we would get if we had an infinite number of processors. To get this value, we had to compute the following equation: $T_{par}/(T_{seq} + T_{par})$.

All these data can be found in *Paraver* or computed by ourselves. In this case, we are not going to use 8 processors but 8 threads.

5.1 Initial version

In this part, we started finding the T_{par} of the execution with one thread using the *OMP_parallel_functions_duration* configuration and the *OMP_state_profile*. These two configurations helped us on it. Then, we just had to sum all the execution times of the different parallelised functions. The results are the following:

$$T_{par} = 1535434.96us = 1535434960ns$$

Afterwards, we wanted to find the sequential time of the program. To do this, we got the total time of the program with one thread. In Figure 23 we can see the trace using only 1 thread. In the lower right corner we can find the total execution time (T_1). Figure 24 shows the same information but in a table. Summing all the values of the "Total" row will result in the same value than the Figure 23.

$$T_1 = 2378316048ns$$

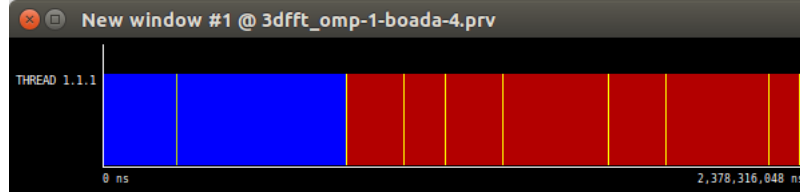


Figure 23: Trace of the execution with 1 thread.

	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,376,650,743 ns	1,190,907 ns	204,572 ns	267,403 ns	2,423 ns
Total	2,376,650,743 ns	1,190,907 ns	204,572 ns	267,403 ns	2,423 ns
Average	2,376,650,743 ns	1,190,907 ns	204,572 ns	267,403 ns	2,423 ns
Maximum	2,376,650,743 ns	1,190,907 ns	204,572 ns	267,403 ns	2,423 ns
Minimum	2,376,650,743 ns	1,190,907 ns	204,572 ns	267,403 ns	2,423 ns
StDev	0 ns	0 ns	0 ns	0 ns	0 ns
Avg/Max	1	1	1	1	1

Figure 24: Table of the diferent states that thread 1 has been.

Then, applying the following formulas, we could find T_{seq} and ϕ .

$$T_{seq} = T_1 - T_{par} = 2378316048 - 1535434.96 * 1000 = 842881088ns$$

$$\phi = \frac{T_{par}}{T_{seq} + T_{par}} = \frac{1535434960}{842881088 + 1535434960} = 0.64559752741$$

$$\phi = 64.559752741\%$$

Then we have changed the trace for the one that has been processed with 8 threads. Figure 25 shows the execution time with 8 threads and the work that each one does.

$$T_8 = 1609711162ns$$

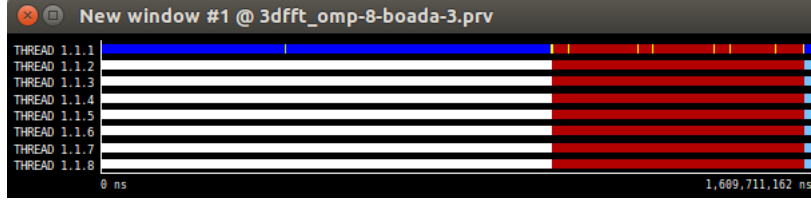


Figure 25: Trace of the execution with 8 threads.

Then, we had to compute S_8 . We calculated this as following:

$$S_8 = \frac{T_1}{T_8} = \frac{2378316048}{1609711162} = 1.47748000023$$

Before calculating S_∞ , we had to compute again T_1 , T_8 and S_8 , but with a different script. This is because *Paraver* is introducing some extra time needed to initialise the instrumentation, but then the behaviour is reflected accurately with no much overhead. Now, executing the script **submit-omp.sh** will not produce this "extra time". We got 3 different values, corresponding to the different parallelised parts: 3D FFT Plan Generation, Init Complex Grid and Execution FFT3D. Summing up all these 3 values will result in T_1 and T_8 depending on the number of processors executing the code.

$$T_1 = 0.005400 + 0.577763 + 1.737352 = 2.320515s = 2320515000ns$$

$$T_8 = 0.000551 + 0.576197 + 0.559041 = 1.135789s = 1135789000ns$$

$$S_8 = \frac{T_1}{T_8} = \frac{2320515000}{1135789000} = 2.0430863479$$

We can observe that there is a big overhead when using *Paraver*. If we do not use it, we increase the speedup a lot. Finally we can compute the S_∞ using the next equation:

$$S_\infty = \frac{1}{1 - \phi} = \frac{1}{1 - 0.64559752741} = 2.8216507427$$

This means that with a 64.55 % of the code parallelised, we will only be able to execute the program 2.82 times faster. The following figures show the evolution of the execution time and the speedup as we increase the number of threads (strong scalability).

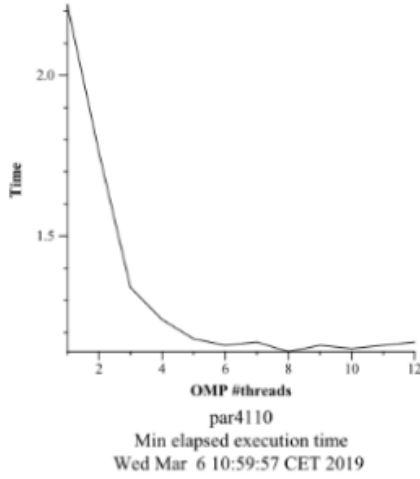


Figure 26: Execution time plot as we increase the number of threads.

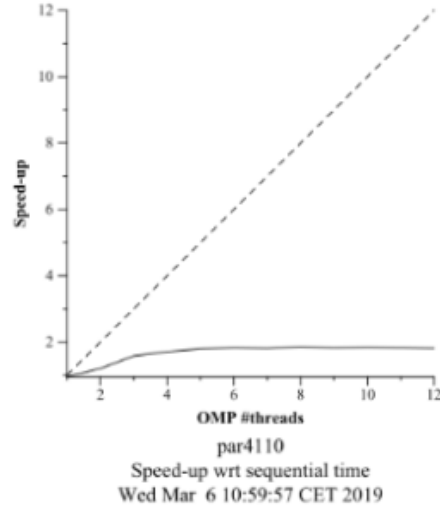


Figure 27: Speedup plot as we increase the number of threads.

5.2 Improving ϕ

In this part of the laboratory, we had to find the part of the code that make our code to be less parallelized. This part was the function `init_complex_grid`. To increase the parallelism we uncommented the pragma shown in the next code.

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
    #pragma omp parallel
    for (int k = 0; k < N; k++)
        #pragma omp for schedule(static, 1)
        for (int j = 0; j < N; j++)
            for (int i = 0; i < N; i++)
            {
                in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)
                    + sin(M_PI*((float)i)/32.0)
                    + sin(M_PI*((float)i)/16.0));
                in_fftw[k][j][i][1] = 0;
            }
    #if TEST
        out_fftw[k][j][i][0] = in_fftw[k][j][i][0];
        out_fftw[k][j][i][1] = in_fftw[k][j][i][1];
    #endif
}
```

Now we had to compute the same equations that we applied in the 1st part in order to find the results for the table. T_1 has not exactly the same value than the previous section. We think that *Paraver* is the culprit.

$$T_1 = 2377989045ns$$

$$T_{par} = 2066229470ns$$

$$T_{seq} = 2377989045 - 2066229470 = 311759575ns$$

$$\phi = \frac{T_{par}}{T_{seq} + T_{par}} = \frac{2066229470}{311759575 + 2066229470} = 0.86889780856$$

$$\phi = 86.889780856\%$$

$$T_8 = 887257925ns$$

$$S_8 = \frac{T_1}{T_8} = \frac{2377989045}{887257925} = 2.68015531673$$

$$S_\infty = \frac{1}{1 - \phi} = \frac{1}{1 - 0.86889780856} = 7.62763756285$$

It makes sense that all the speedup values have increased and the execution times have decreased, because as we have augmented the parallelism, more threads can now execute code at the same time. Some of the previous values have been extracted from the following figures:

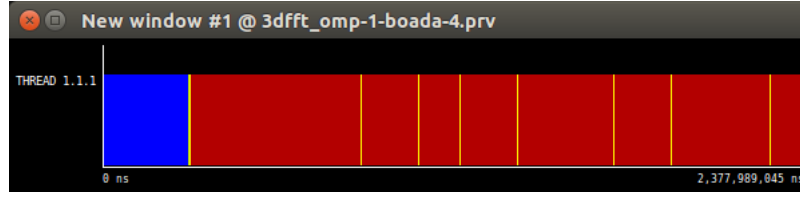


Figure 28: Trace of the execution with 1 thread.

	[103,905..106,393]	[141,233..143,722]	[191,004..193,492]	[193,492..195,981]	[327,874..330,363]	[330,363..332,851]	[576,729..579,217]
THREAD 1.1.1	103,904.79 us	142,715.92 us	192,723.17 us	388,080.83 us	328,836.91 us	332,054.19 us	577,913.66 us
Total	103,904.79 us	142,715.92 us	192,723.17 us	388,080.83 us	328,836.91 us	332,054.19 us	577,913.66 us
Average	103,904.79 us	142,715.92 us	192,723.17 us	388,080.83 us	328,836.91 us	332,054.19 us	577,913.66 us
Maximum	103,904.79 us	142,715.92 us	192,723.17 us	388,080.83 us	328,836.91 us	332,054.19 us	577,913.66 us
Minimum	103,904.79 us	142,715.92 us	192,723.17 us	388,080.83 us	328,836.91 us	332,054.19 us	577,913.66 us
StDev	0 us	0 us	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1	1	1

Figure 29: Table of the times of the different parallelised functions.

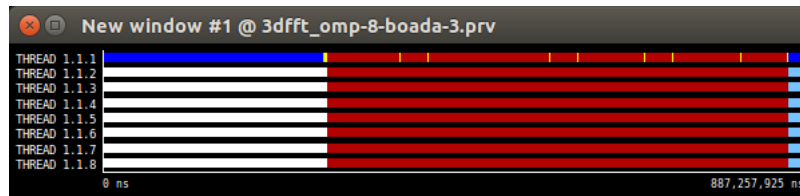


Figure 30: Trace of the execution with 8 threads.

Finally, in figures 31 and 32 we can see that the speedup increases more than the figure of the previous section and the time starts to increase again when the number of threads is equal to 10 (in the previous section it started in 8 threads).

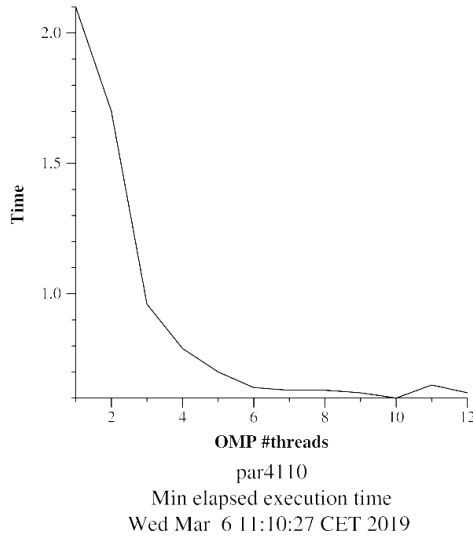


Figure 31: Execution time plot as we increase the number of threads.

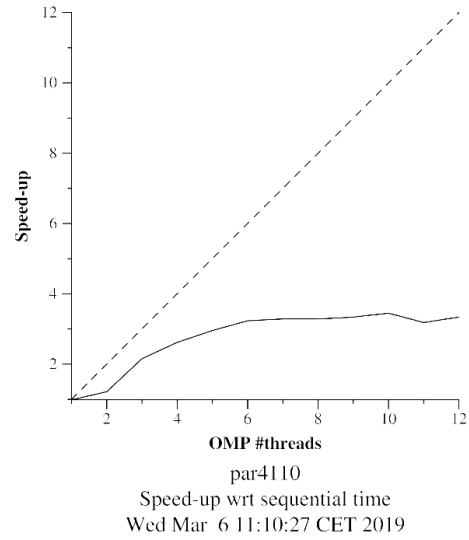


Figure 32: Speedup plot as we increase the number of threads.

5.3 Reducing parallelisation overheads

Lastly, we had to increase the granularity of tasks by moving one line up the second pragma of each function. For example:

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
    #pragma omp parallel
    #pragma omp for schedule(static, 1)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            for (int i = 0; i < N; i++)
            {
                in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)
                    + sin(M_PI*((float)i)/32.0)
                    + sin(M_PI*((float)i)/16.0));
                in_fftw[k][j][i][1] = 0;

                out_fftw[k][j][i][0] = in_fftw[k][j][i][0];
                out_fftw[k][j][i][1] = in_fftw[k][j][i][1];
            }
    #if TEST
    #endif
}
```

This change allowed us to reduce the parallelization overheads getting a better performance as seen in the following equations (the same that are explained in the 1st part).

$$T_1 = 2349744097ns$$

$$T_{par} = 2081319010ns$$

$$T_{seq} = 2349744097 - 2081319010 = 268425087ns$$

$$\phi = \frac{T_{par}}{T_{seq} + T_{par}} = \frac{2081319010}{268425087 + 2081319010} = 0.88576411901$$

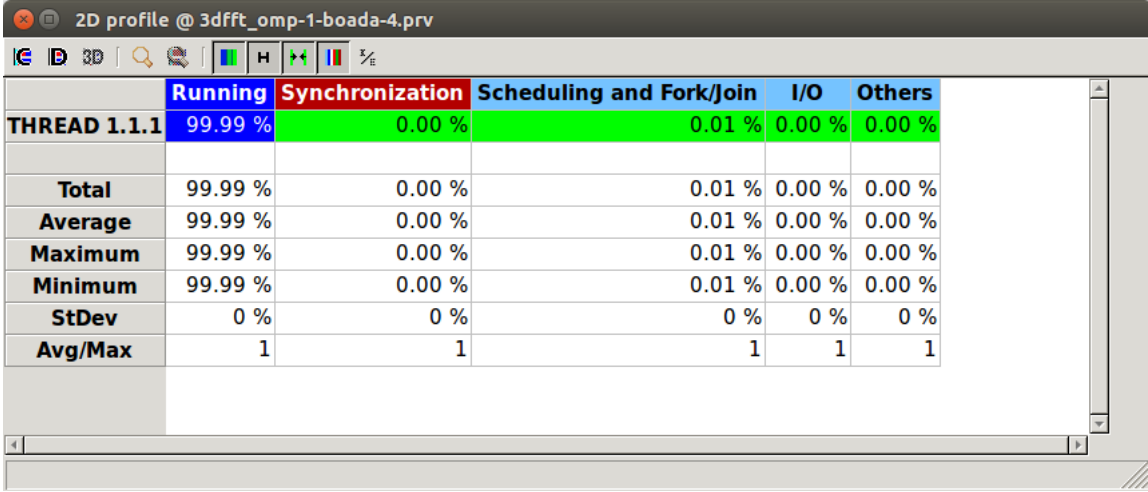
$$\phi = 88.576411901\%$$

$$T_8 = 639450975ns$$

$$S_8 = \frac{T_1}{T_8} = \frac{2349744097}{639450975} = 3.67462743645$$

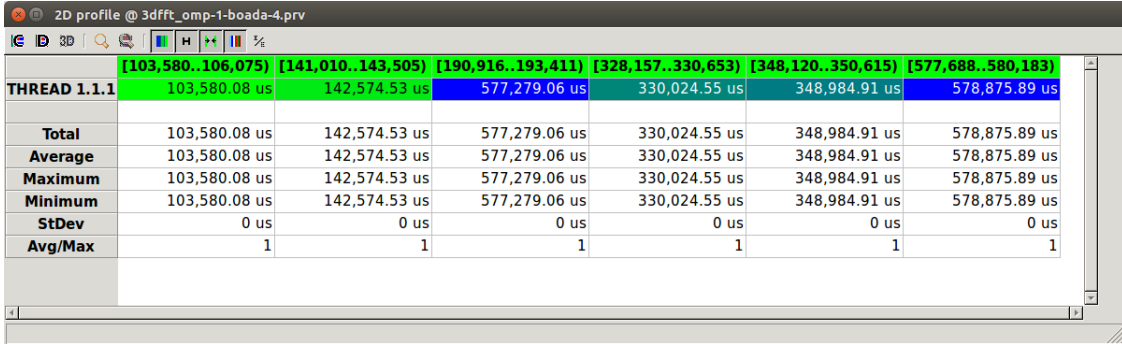
$$S_\infty = \frac{1}{1 - \phi} = \frac{1}{1 - 0.88576411901} = 8.75381702608$$

We can see that this change made us to improve the S_8 (2.68 to 3.67) and S_∞ (7.62 to 8.75). Following we show some of the traces and tables we have used to compute all these values.



	Running	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.99 %	0.00 %	0.01 %	0.00 %	0.00 %
Total	99.99 %	0.00 %	0.01 %	0.00 %	0.00 %
Average	99.99 %	0.00 %	0.01 %	0.00 %	0.00 %
Maximum	99.99 %	0.00 %	0.01 %	0.00 %	0.00 %
Minimum	99.99 %	0.00 %	0.01 %	0.00 %	0.00 %
StDev	0 %	0 %	0 %	0 %	0 %
Avg/Max	1	1	1	1	1

Figure 33: Table of the diferent states that thread 1 has been.



	[103,580..106,075]	[141,010..143,505]	[190,916..193,411]	[328,157..330,653]	[348,120..350,615]	[577,688..580,183]
THREAD 1.1.1	103,580.08 us	142,574.53 us	577,279.06 us	330,024.55 us	348,984.91 us	578,875.89 us
Total	103,580.08 us	142,574.53 us	577,279.06 us	330,024.55 us	348,984.91 us	578,875.89 us
Average	103,580.08 us	142,574.53 us	577,279.06 us	330,024.55 us	348,984.91 us	578,875.89 us
Maximum	103,580.08 us	142,574.53 us	577,279.06 us	330,024.55 us	348,984.91 us	578,875.89 us
Minimum	103,580.08 us	142,574.53 us	577,279.06 us	330,024.55 us	348,984.91 us	578,875.89 us
StDev	0 us	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1	1

Figure 34: Table of the times of the different parallelised functions.

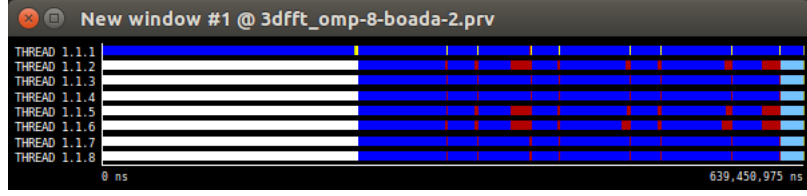


Figure 35: Trace of the execution with 8 threads.

Figures 36 and 37 show the improvement we achieved. The speedup is the biggest between the three codes and the time still decreases when we have 10 threads.

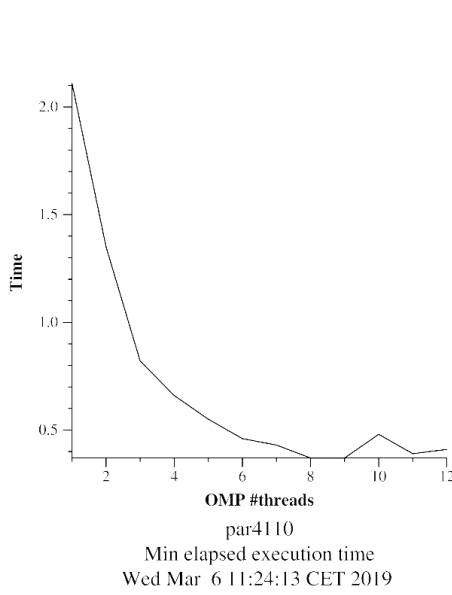


Figure 36: Execution time plot as we increase the number of threads.

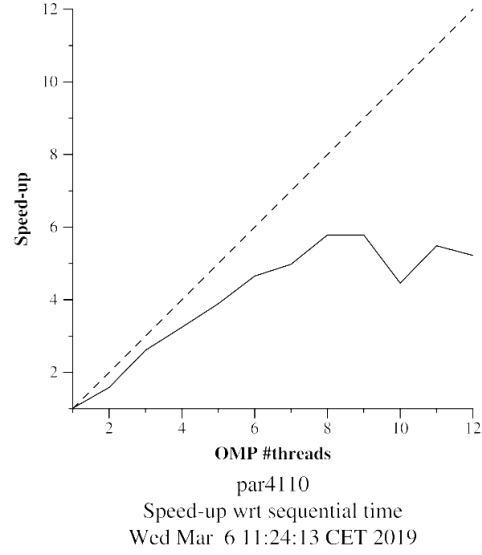


Figure 37: Speedup execution time plot as we increase the number of threads.

5.4 Conclusions

Version	ϕ (%)	S_{∞}	T_1 (s)	T_8 (s)	S_8
initial version in 3dfft_omp.c	64.55	2.82	2.38	1.61	1.48
new version with improved ϕ	86.89	7.63	2.38	0.89	2.68
final version with reduced parallelisation overheads	88.58	8.75	2.35	0.64	3.67

Table 7: Analysis of execution time of 3dfft program.

In this code, we have been able to observe the temporary improvement in the evolution of the code. First of all, when we added parallelism in the first function there has been a great change especially in the scalability, since much of the sequential code has been part of the parallel. Next, by increasing the granularity of the tasks, we have seen a small improvement, especially in the parallel part, since we have been able to reduce overheads.